Automating Object Transformations for Dynamic Software Updating

Stephen Magill *
IDA Center for Computing Sciences
sbmagil@super.org

Michael Hicks
University of Maryland, College Park
mwh@cs.umd.edu

Suriya Subramanian
Intel Corporation
suriya@alumni.cs.utexas.edu

Kathryn S. McKinley
Microsoft Research & The University of Texas at Austin
mckinley@cs.utexas.edu

Abstract

Dynamic software updating (DSU) systems eliminate costly downtime by dynamically fixing bugs and adding features to executing programs. Given a static *code* patch, most DSU systems construct runtime code changes automatically. However, a dynamic update must also specify how to change the running program's execution *state*, e.g., the stack and heap, to make it compatible with the new code. Constructing such *state transformations* correctly and automatically remains an open problem. This paper presents a solution called *Targeted Object Synthesis* (TOS). TOS first executes the same tests on the old and new program versions separately, observing the program heap state at a few corresponding points. Given two corresponding heap states, TOS *matches* objects in the two versions using *key* fields that uniquely identify objects and correlate old and new-version objects. Given example object pairs, TOS then *synthesizes* the simplest-possible function that transforms an old-version object to its new-version counterpart. We show that TOS is effective on updates to four open-source server programs for which it generates non-trivial transformation functions that use conditionals, operate on collections, and fix memory leaks. These transformations help programmers understand their changes and apply dynamic software updates.

* Most of the work was completed while this author was at the University of Maryland, College Park.

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; I.2.2 [*Artificial Intelligence*]: Program Synthesis

General Terms Algorithms, Languages, Theory

Keywords Dynamic Software Update, DSU, Hot-Swapping, Program Synthesis, State Transformation, Object Correlation, Object Matching

1. Introduction

Suppose you are running an on-line service and a memory leak in your server software causes it to regularly run out of memory and crash. Eventually you discover the one-line fix: the `Connection` class's `close` method should unlink some metadata when a connection closes. To apply this fix in a standard deployment, you stop your server and restart the patched version, but this disrupts active users. With *dynamic software updating* (DSU) support in an extended Virtual Machine such as LiveRebel [16] you can do better. You apply a *dynamic* patch to the `Connection` class of your *running* system to prevent further leaks without disrupting current users. In some DSU-enhanced VMs, such as Jvolve [14], you can do better still. You include *state transformation* code in the dynamic patch that traverses the heap and unlinks the useless metadata left reachable by the bug.

An important goal toward furthering the adoption of DSU systems is to make them easy to use, i.e., to minimize the effort required to produce a correct dynamic patch from two versions of a system. As a step in this direction, many DSU systems employ simple *syntactic*, *type-based* tool support for constructing a dynamic patch from the old and new program versions [1, 8, 13, 14]. For example, if the bytecode for method `m` of class `C` changes, Jvolve will include `C.m` in the dynamic patch. If `C`'s field definitions change, in type or number, Jvolve creates a default *object transformation function* that it applies to all `C` objects when it applies the

patch. This function retains the values of unchanged fields and initializes the rest with a default value, e.g., **null**.

While tool support for identifying changed code is highly effective, existing support for constructing state transformation code is rarely sufficient, and the programmer must therefore modify the generated code. For example, in Jvolve the programmer must add code that unlinks the leaked metadata in our example. Unfortunately, the cases that require manual intervention are often challenging to get right. For the above example, the Connection transformer cannot simply unlink all connection metadata unconditionally. Instead, it must use appropriate context by examining the running program's heap and stack to identify and unlink only the metadata that is logically dead. Transformations that move objects between collections or partition single objects into several objects—examples we observe in practice—require similar care in their construction. Thus, writing state transformation code for DSU systems is a programming task unique to DSU, and it can be a time-consuming, error-prone process.

To ease this burden on programmers, we have developed a general-purpose approach for synthesizing object transformers that we call *Targeted Object Synthesis* (TOS). Our development is in the context of a DSU system for Java, called Jvolve, but our techniques are readily adaptable to other DSU systems. Furthermore, the techniques that we design for finding heap object correlations between different program versions may be useful for other program understanding tasks, such as bug detection and testing.

TOS works in two phases, *matching* and *synthesis*—the matching phase creates examples by pairing objects in two snapshots taken at equivalent points during the execution of the old and new programs, respectively, while the synthesis phase generates a function that transforms the old object of a matched example pair to the new object.

The matching phase begins by running both the old and new versions of the program on the same inputs and taking heap snapshots at corresponding program points. Given a class C whose fields changed between versions, we first reduce the heap snapshots such that they only include C objects and objects to which they refer, directly or transitively. TOS matching seeks to correlate objects in the old and new versions. TOS identifies *key fields* in the objects that (1) uniquely identify the object in a given heap (i.e., each object of class C differs on the values of its key fields) and (2) there exist objects in both heaps with the same values for these fields. We use a greedy algorithm which, in our experience, usually succeeds in finding a set of key fields. In the case that no key fields exist, matching uses the most distinguishing set of fields it can find to pair up most objects, and then applies a lightweight form of synthesis to find a function that pairs up the remaining objects.

With example pairs of corresponding old o and new o' objects (o, o') in hand, the *synthesis* phase searches for functions that are consistent with the examples, i.e., functions δ for which $\delta(o) = o'$ for all matched pairs. Functions δ assign an expression to each new-version object field one at a time, where the expressions may reference any of the old object's fields (or fields reachable from them). These expressions may contain constants, simple functions (e.g., the concatenation or partitioning of string expressions), and conditionals (e.g., if the value to assign to a field depends on the current value of another field). While these expression forms are sufficient for our examples, additional expression forms can be readily supported, expanding expressiveness at the cost of increasing the search space. For collections, we recursively invoke synthesis to generate transformations between objects that make up each collection, mapping the resulting function over the old collection objects to produce the new one. When many functions are possible for a given set of examples, synthesis chooses the simplest. We carefully designed the transformation language to make important operations efficient, such as intersecting a set of candidate functions.

As far as we are aware, the TOS matching algorithm is new. No prior work attempts to map heap objects from unstructured heap snapshots of different program-version executions. The TOS synthesis algorithm is inspired by recent work on synthesizing string and Excel table data transformation functions from input and output examples [5, 6]. TOS matching creates examples *automatically*, whereas this prior work requires users to provide examples. TOS functions are a superset of string transformations. Whereas Excel table functions focus on filters and numerics, TOS data transformations focus on a more general problem, the transformation of heap objects.

We demonstrate our approach by synthesizing transformation functions for updates to several open-source Java servers, including JavaEmailServer (a POP and SMTP server), CrossFTP (an FTP server), Azureus (a Bittorrent client), and JEdit (a graphical text editor). In one case, changed objects do not have the key fields that matching requires. In all the others, we show that TOS produces correct transformation functions. These functions include object field additions, string partitioning, partitioning a collection based on a predicate, and deleting objects due to memory leaks. In fact, to our knowledge no prior DSU system has considered the need to correct the residual effects of bugs, such as a memory leak, and our synthesized functions are the first demonstration of this capability. TOS represents a substantial step toward realizing the promise of DSU technology by reliably automating the most programmer-intensive step.

2. Overview

This section presents an overview of synthesizing state transformation functions for dynamic software updates using TOS. We begin with some background on DSU, present an example dynamic update taken from an actual program

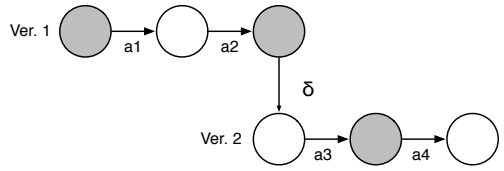

Figure 1. Program trace at update point after a2.

change, and show how TOS synthesizes this dynamic update automatically.

2.1 Dynamic software updating

Suppose an old version of a program is actively running, and a new version becomes available that fixes some bugs or adds some new features. In many cases, we would like to update the running program without shutting it down since stopping it would degrade the user experience or the program contains useful program state that is costly to recreate. For example, users may have active connections or the program may cache state, such as recent queries and network state.

To use a typical DSU system, we must construct a *dynamic patch* [8] that specifies the changed code and a *state transformation function*, which modifies heap objects and other program state, as necessary, to work with the new code. For example, if the old program version maintains a list of Connection objects and the new version adds some fields to the Connection class, the state transformation function must initialize the values of the new fields for the existing objects. In some systems, the state transformer may also update the *control state* of the program, e.g., examining and modifying the existing stack and program counter as necessary [10, 14]. We implement TOS for Jvolve [14], which performs DSU in a Java Virtual Machine (Jikes RVM), but the TOS design generalizes to other DSU systems.

Figure 1 depicts a dynamically updated program's execution. The circles represent the program's state, the labels a_1, a_2, a_3, a_4 represent *actions*, e.g., messages sent to and from client applications. Each gray circle represents a state in which a dynamic update is permitted—not every program state may be amenable to certain dynamic updates, as discussed below. In this trace, the program starts executing at version 1, and after executing actions a_1 and a_2, it applies a dynamic patch. As a result, the code of the program is updated to version 2, and the state transformation function δ is applied to transform the current state. A patch could have been applied in the initial state or the one after a_3, if a patch were available at those times.

Most DSU systems work in three steps. First, when a patch becomes available and the program reaches an acceptable state, the DSU system dynamically loads the new and changed code. Second, it redirects existing references to the new definitions. Finally, it executes the state transformation function to update the existing state. Jvolve implements these steps within a modified virtual machine. It uses standard classloading to load new versions of classes. For classes whose only change is to the code of methods, Jvolve simply modifies the metadata for that class to point to the new method definitions (which the JIT may subsequently optimize). For each class whose objects' state requires modification (e.g., because the new version adds fields), the patch must include an *object transformation method*. At update time, the garbage collector finds all objects that require transformation. It executes the object transformation method on each old object, creating and initializing a corresponding object that conforms to the new class's type specification.

```
public class v131_User {
  private final String username, domain, password;
  private String[] forwardAddresses;
}
public class JvolveTransformers {
  ...
 public static void
  jvolveObject(User n, v131_User o) {
    n.username = o.username;
    n.domain = o.domain;
    n.password = o.password;
    int len = o.forwardAddresses.length;
    n.forwardAddresses = new EmailAddress[len];
    for (int i = 0; i < len; i++) {
      String[] parts = o.forwardAddresses[i].split("@");
      n.forwardAddresses[i] = new EmailAddress(parts[0], parts[1]);
}}}
```

Figure 2. User object transformer, JES 1.3.1–1.3.2 update

When a dynamic patch becomes available, the system may choose not to apply it immediately. A policy adopted by many DSU systems is to delay updates while changed code is actually executing or referenced by the call stack. While this delay makes sense, it is not sufficient to avoid trouble. Hayden et al. [7] studied several years' worth of changes to three server programs and found that dynamic updates derived from actual releases sometimes fail even while adhering to this "activeness" restriction. Other work [8] suggests that simply asking programmers to specify a few program points (dubbed *update points*) at which updates are permitted makes the system easier to reason about. Hayden et al.'s study finds this approach to be effective: updates were applied promptly (e.g., roughly every 10 ms) and never failed. Jvolve and several other systems [8, 10, 13] support this approach. TOS uses update points to create correlated pairs of heap snapshots, as explained in Section 2.4.

2.2 JavaEmailServer example

We now present an example Jvolve dynamic update and subsequently show how TOS synthesizes it. Figure 3 illustrates code from versions 1.3.1 and 1.3.2. of JavaEmailServer (JES), a simple SMTP and POP e-mail server, that we obtained from the JES open-source repository. In the old

```
public class User {
  private final String username, domain, password;
  private String [] forwardAddresses;
  public User (...) {...}
  public String [] getForwardedAddresses() {...}
  public void setForwardedAddresses(String[] f)
  {...}
}
public class ConfigurationManager {
  private User loadUser(...) {
      ...
      User user = new User(...);
      String [] f = ...;
      user.setForwardedAddresses(f);
      return user;
  }
}
```

(a) Version 1.3.1

```
public class User {
  private final String username, domain, password;
  private EmailAddress[] forwardAddresses;
  public User (...) {...}
  public EmailAddress[] getForwardedAddresses() {...}
  public void setForwardedAddresses(EmailAddress[] f)
  {...}
}
public class ConfigurationManager {
  private User loadUser(...) {
      ...
      User user = new User(...);
      EmailAddress[] f = ...;
      user.setForwardedAddresses(f);
      return user;
  }
}
public class EmailAddress {
  public EmailAddress(String username, String domain) {
    _isEmpty = false;
    _username = username;
    _domain = domain;
  }
  ...
  private String _username = "";
  private String _domain = "";
  private boolean _isEmpty = true;
}
```

(b) Version 1.3.2

Figure 3. An update to JavaEmailServer (JES) User and ConfigurationManager classes

version of the User class, forwardedAddresses is an array of strings. In the new version, forwardedAddresses is an array of EmailAddress objects. This difference requires a corresponding change to the types of other methods in the User class, and to the loadUser method code of the ConfigurationManager class, which sets the field by calling setForwardedAddresses.

A Jvolve dynamic patch for this update contains the new versions of the User and ConfigurationManager classes. No object transformer is needed for the ConfigurationManager class because only its methods have changed, not its fields. Figure 2 illustrates the object transformer method for User objects. The transformer is a **static** method jvolveObject in the class JvolveTransformers. The method takes the old-version object and an allocated uninitialized new-version object as arguments. Both have the same class name. To distinguish them, Jvolve renames the old object's class to v131_User. The transformation method copies the first three fields from the old to the new version.[1] The object transformer allocates and populates an array of EmailAddress objects to replace the existing array of String objects.

[1] Jvolve relaxes the Java language restrictions on private field accesses during an update.

Given two program versions, Jvolve and other DSU systems automatically construct the code portion of a dynamic patch by syntactically comparing the old and new class files. However, generating the object transformer in Figure 2 is well beyond the reach of current techniques. Jvolve produces the first three lines, but then inserts the line n.forwardedAddresses = **null**. Ginseng [13], POLUS [4], and DLpop [8] do slightly better: they generate the loop, but the loop body simply assigns each element to null. TOS generates the correct object transformer for JES in its entirety.

2.3 Snapshot Collection

TOS infers object transformers based on example pairs of (old-version, new-version) objects and uses the test cases already present in a programs test suite to produce these examples. It does this by executing each test case twice—once with the old version and once with the new version of the code. At every update point encountered during execution, it records a *heap snapshot*, which records types and field values for all live objects. Figure 4 depicts this process. The shaded circles are the update points at which we take heap snapshots. Thus each version in the figure will produce three snapshots and these will later be compared to find example pairs for synthesis.

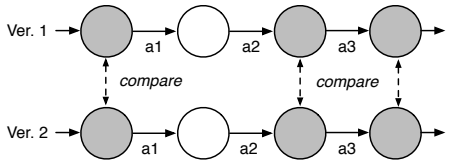

Figure 4. Comparing old and new heaps at update points.

Snapshot collection is generally straightforward once the programmer marks the update points (which is already required for dynamic updating). Even far-reaching, complex changes to code can be accommodated since update points tend to be close to the root of the control-flow graph, in code that is quite stable. For example, our JavaEmailServer code has an update point at the end of the main processing loop for the thread that handles sending outgoing messages. If a test case involves sending specific messages at specific intervals, then the same number of snapshots will be created during each run. Futhermore, these snapshots will correspond in the sense that the lists of sent and pending messages will be the same. Changes to the send routine, or the details of the protocol used to send messages, do not affect this correspondence. Only updates that alter the high-level message processing semantics are problematic—for example, if the server switched from sending all pending messages at once to a model where message sends are spaced out (perhaps to implement a rate-limited send). Such changes require great care because an existing invariant—that there are no pending messages when the update point is reached—no longer holds in new program versions. These updates are best left to the programmer, so we do not view the failure of TOS to cope with them as a significant shortcoming.

2.4 Matching

TOS works in two steps, *matching* and *synthesis*. Matching takes as input the snapshots produced by the test runs and the class C for which we want to generate an object transformer. It then produces pairs of (old-version, new-version) objects that serve as examples to guide synthesis. Early in the matching process we also prune each snapshot to include only C objects and objects to which they refer, directly or transitively. This linear pass through the heap significantly reduces the input size to TOS matching.

In general, TOS requires that (1) the test input generate the same number of snapshots when run using each program version; and (2) the i^{th} snapshot for a given input will always contain the same set of objects of class C; and (3) the corresponding snapshots have the same number of C objects (though the number of instances of other classes may differ). Intuitively, the program must behave deterministically with respect to the objects in C, and the *role* of those objects in the two versions must be the same. The JES example illustrates this point. The email server itself is non-deterministic. Network events and the order in which requests are processed vary from one run to the next. However the set of forwarded addresses behaves deterministically since it is read from the same configuration file in both versions and does not vary across runs. For all our real-world applications and test inputs, these requirements were met.

The goal of the *matching* phase is to produce a uniform one-to-one mapping between objects in each pair of corresponding snapshots. It does this primarily by identifying a class's key fields.

Definition The fields $\vec{f}$ of a class C are *key fields* if objects of class C have two properties.

1. No pair of C objects in the same snapshot have the same values for all the fields $\vec{f}$.
2. For each object in snapshot σ_{old}, there is exactly one object in the corresponding snapshot σ_{new} that has the same values for fields $\vec{f}$.

As an example, Figure 5 illustrates a snapshot pair from JES with three objects of class User in the old (top) and new (bottom) snapshots. In this case, TOS generates the mapping $\{(\text{user[0]}_{old}, \text{user[0]}_{new}), (\text{user[1]}_{old}, \text{user[1]}_{new}), (\text{user[2]}_{old}, \text{user[2]}_{new})\}$ using key field username. This field is key because the values john, alice, and pat uniquely identify each object in both σ_{old} and σ_{new} and there exists only one old and new object with the same value. Neither domain nor password are key fields because multiple objects in the same snapshot have the same value.

If user[2].username was john instead of pat, then there is no *single* key field, since none of the primitive fields username, domain, or password have unique values. In this case, matching searches for a set of fields that all together satisfy the given criteria. Given the modified user[2] snapshots, matching would find that together the fields username and domain impose a one-to-one mapping on all the objects.

If no set of key fields exists, our matching algorithm employs two refinements, described in Section 3. First, it attempts to use referent object field values as potential key fields. If this attempt fails, it uses the set of key fields/paths that is most discriminatory (i.e., it comes the closest to producing a one-to-one matching), and then refines any objects not yet matched by using a lightweight form of synthesis.

2.5 Synthesis

The matching phase ultimately produces a list of example pairs of objects of a class C. Synthesis then proceeds in roughly two steps. (1) For each example pair, the algorithm synthesizes a set of candidate functions. (2) It intersects the set of candidate functions to produce a function consistent with all examples. In our implementation, the algorithm proceeds by synthesizing an initializer for each field, one at a time. The description given here, for simplicity, presents the process as working for the entire transformer method at once.

Old Version Heap Objects JES Heap		
user[0]	**user[1]**	**user[2]**
john	alice	pat
yahoo.com	yahoo.com	intel .com
poorpassword	poorpassword	poorpassword
["john@cs.umd.edu", "john−alice@yahoo.com"]	["john−alice@yahoo.com"]	NULL
New Version Heap Objects JES Heap		
user[0]	**user[1]**	**user[2]**
john	alice	pat
yahoo.com	yahoo.com	intel .com
poorpassword	poorpassword	poorpassword
forwardedAddresses[0] =	forwardedAddresses[0] =	NULL
[_username = "john", _domain = "cs.umd.edu"]	[_username = "john−alice",	
forwardedAddresses[1] =	_domain = "yahoo.com"]	
[_username = "john−alice", _domain = "yahoo.com"]		

Figure 5. JES heap example for User Class with fields, in order, username, domain, password, and forwardedAddresses.

The first step proceeds as follows. For each example pair, synthesis seeks a *set* of functions Δ such that each $\delta \in \Delta$ is consistent with the example, i.e., $o' = \delta(o)$ for the example (o, o'). Each $\delta_i \in \Delta$ assigns each new-version field one at a time. For example, consider the two User objects in Figure 5 that correspond to user[0]. For this example, synthesis will first infer δ_0 that assigns the *constants* john, yahoo.com, and poorpassword to each of the new object's fields username, domain, and password, respectively. It also infers δ_1 that *copies* the corresponding fields from the input object, i.e., n.username := o.username, n.domain := o.domain, etc. For fields of type String, it also considers assigning the concatenation of other strings, e.g., those that are substrings of old-version fields.

For fields that are collections, we invoke synthesis recursively. The algorithm matches two collections and then matches the set of objects in the two collections. It then generates a transformation function between the collection pairs from a transformation function for the object pairs. For forwardedAddresses, each String has the form "x@y" and is mapped to an EmailAddress object whose username, domain, and _isEmpty fields are "x", "y", and **false**, respectively, where the first two fields are substrings of the input string. Once the algorithm establishes an element-wise function, it simply iterates over the old collection and maps each element to one in the new collection.

Once synthesis generates Δ for each pair of objects of class C, it intersects them to produce a $\hat{\Delta}$ that is consistent with all the examples. During this step, synthesis discards overly-specific functions, e.g., it discovers that δ_1 described above is consistent with all three of the matched pairs but δ_0 is not, and discards it. If $|\hat{\Delta}| = 1$, then synthesis chooses $\delta \in \hat{\Delta}$ for the object transformer. If $|\hat{\Delta}| > 1$, it picks the $\delta \in \Delta$ that is intuitively the *simplest* and *most general* function. For example, we mark functions that contain assignments from old fields to new fields as more general than functions that assign constants. If $|\hat{\Delta}| = 0$ then no one function works for all examples. In this case, synthesis picks the function that works for the most examples and then iteratively seeks a function that works for the remaining examples along with a conditional expression that distinguishes between the two cases.

2.6 Discussion

A key difference between TOS and prior work on learning from examples is that in prior work, the user identifies particular example pairs, whereas in our work, the user must produce matching *executions*—created by running the old and new program versions on the same inputs—from which TOS automatically identifies examples. While automatic matching is simpler than identifying example object pairs directly, there is a risk that it will not pair up truly corresponding objects, in which case the synthesized transformation function will be incorrect. To reduce the likelihood of this case, we place strong restrictions on the inputs to matching, as described in the third paragraph of Section 2.4: snapshots in both versions must be taken at the same update points in both versions, there must be an equal number of snapshots, and when synthesizing a transformer for class C, there must be an equal number of C objects in corresponding snapshots. These restrictions ensure that C objects are playing the same role in the old and new execution, so if we update the old program at one of these update points, executing the synthesized transformer would bring the program to an equivalent state.

Note that while corresponding snapshots must contain the same number of C objects, where C is the changed class, the number of other objects can vary. For example, a buggy implementation of C may fail to null its field f of class D, inducing a memory leak. Thus in the old and new snapshots, the number of C objects will be the same, but the number

of D objects may differ. As we show in our experiments, inferring the transformer method for such a leaky class C can end up correcting the memory leak by nulling the dead objects.

Because TOS generates a solution specific to the examples it is given, the developer must snapshot executions that produce a sufficient number of objects. For conditional transformers, the input must produce objects that cover the range of possible variations; i.e., if there are N conditions, TOS needs at least an example pair for each of the N conditions. These examples could come from a single snapshot with N instances of the object in both the old and new version or from multiple snapshots which each contain one or two instances but together provide N examples. If there are too few examples, then synthesis may infer a function that is overly specific.

In our experience (described in Section 5), it was easy to provide enough examples for synthesis, and most experiments required just one (well-chosen) test. TOS fails in two of our test cases. In one case we could not reproduce a memory leak we were aiming to fix, leading to a failure during the snapshot collection phase. In the other case, the changed objects did not have key fields and so the matching phase failed. Despite these limitations, our approach adds value: the developer needs to run tests anyway, and if these tests are sufficiently deterministic and cover the relevant behaviors of a changed class, TOS can be used to infer object transformers for that class.

3. Matching

Now we present the TOS matching algorithm in detail; the next section describes the synthesis algorithm. The goal of matching is to produce example pairs (o, o') of corresponding old and new objects taken from heap snapshot pairs. The synthesis phase takes these pairs as input and searches for a function δ such that $\delta(o) = o'$ for all the example pairs. The functions are class based, thus all old objects o must have the same class C and all new objects must have the same class C'. We first assume $C = C'$, and then consider $C \neq C'$, when matching recursively during synthesis.

We describe our algorithm using the following notation. A snapshot σ is just a set of objects; we use the two terms interchangeably. We write $\vec{X}$ to denote a list of Xs; $\vec{X} :: X$ to denote concatenating the element X to the end of the list $\vec{X}$; and $\vec{X}(i)$ to denote the i^{th} element of the list $\vec{X}$. We sometimes refer to a list as a *tuple* (e.g., when its length is known to be fixed). We write $o.\vec{f}$ to denote the tuple $\vec{v}$ where $o.f_i = v_i$ for $0 < i \leq n$. We write $values_{\vec{f}}(\sigma)$ for the set of value tuples assigned to fields $\vec{f}$ by objects in σ, i.e., $values_{\vec{f}}(\sigma) = \{\vec{v} \mid o \in \sigma \wedge o.\vec{f} = \vec{v}\}$. Finally, we write $\vec{\sigma}\lfloor_{\vec{f}=\vec{v}}$ for $\{o \mid o \in \sigma \wedge o.\vec{f} = \vec{v}\}$.

Figure 6 gives the pseudocode for the matching algorithm in the function called match. The input to match is a pair of lists of object sets $(\vec{\sigma}_{old}, \vec{\sigma}_{new})$. The object set $\vec{\sigma}_{old}(i)$ contains objects collected from the i^{th} snapshot taken while running the old program, while $\sigma' = \vec{\sigma}_{new}(i)$ contains objects collected from the corresponding snapshot of a run of the new program.

As mentioned earlier, we first *prune* the snapshots to include only objects of classes C that changed between the old and new version and objects to which these objects directly or transitively refer. We assume that this pruning has happened prior to the call to match. We also assume that there are the same number of snapshots for the old and new program executions ($|\vec{\sigma}_{old}| = |\vec{\sigma}_{new}|$), and that each corresponding pruned snapshot has the same number of objects ($|\vec{\sigma}_{old}(i)| = |\vec{\sigma}_{new}(i)|$). If these conditions do not hold then matching fails for the class in question. The match function returns a list of object pairs (o, o'), the first from an old snapshot and the second from a new snapshot, which serves as input for synthesis.

3.1 Key fields

The match function first calls get_keyfields (line 2) to search for key fields that partition the objects in corresponding pruned snapshots. Given a list of fields kfs, and the old and new snapshots, match calls the function split_on_keyfields, which returns a pair of object set lists whose i^{th} elements correspond. In particular, each object $o \in \vec{\sigma}_{old}(i)$ and $o' \in \vec{\sigma}_{new}(i)$ have the same values for fields in kfs. Ideally, the size of $\vec{\sigma}_{old}(i)$ and $\vec{\sigma}_{new}(i)$ returned by split_on_keyfields will be 1 for all i. In this case (as checked on line 4), each object is uniquely identified by the fields kfs in every snapshot and there is a corresponding object in the old (respectively, new) snapshot with the same field values. The size of a set will be greater than 1 if there are multiple objects in a single snapshot that contain the same values for fields in kfs. If sets are non-singleton, match uses lightweight synthesis to complete the partition.

The function get_keyfields iteratively adds new fields to the list kfs. When it reaches a fixed point, it returns the list. The first nested loop (line 12) considers each possible relevant field f for objects of class τ. The function assigns each field a *score* based on how well f, when added to the current fields in kfs', distinguishes the objects. The inner loop (line 15) considers each pair of snapshots and computes the set V, which contains the distinct tuples of kfs' fields' values in old-version objects. Line 17 then considers each of the tuples $\vec{v}$ in set V. It must be the case that we have the same number of old and new objects with values $\vec{v}$ in fields kfs'. This requirement preserves the ability to discover a bijection between the objects. The larger $|V|$ is, the finer the partition induced by splitting on kfs'. Synthesis prefers finer partitions, which indicate more effective key fields. Thus match adds $|V|$ to score and then chooses the field that maximizes score. If a field f leads to the condition on line 17 being violated, then match assigns f a score of 0 and proceeds to the next field.

```
1  match(σ⃗_old, σ⃗_new) =
2    kfs := get_keyfields (σ⃗_old, σ⃗_new)
3    (σ⃗'_old, σ⃗'_new) = split_on_keyfields(kfs,σ⃗_old,σ⃗_new)
4    if ∃i. |σ⃗'_old(i)| > 1 then
5      (σ⃗'_old, σ⃗'_new) := synthesis_match(σ⃗'_old, σ⃗'_new)
6    return {(o,o') | ∃i.0 < i ≤ length(σ⃗'_new) ∧ o ∈ σ⃗'_old(i) ∧ o' ∈ σ⃗'_new(i)}

7  get_keyfields (σ⃗_old, σ⃗_new) =
8    kfs := []
9    currscore := 0
10   repeat {
11     prevkfs := kfs;
12     for each field f ∉ kfs {
13       kfs' := kfs :: f
14       score(f) = 0
15       for each i ∈ 1..length(σ⃗_new) {
16         V := values_kfs'(σ⃗_old(i))
17         if ∀v⃗ ∈V. |σ_old(i)|_kfs'=v⃗| = |σ_new(i)|_kfs'=v⃗| then
18           score(f) := score(f) + |V|
19         else {
20           score(f) = 0
21           break
22       } }
23     }
24     let g be the f that maximizes score(f)
25     if score(g) > currscore {
26       kfs := kfs :: g
27       currscore := score(g)
28     }
29   } until (prevkfs = kfs)
30   return kfs
31 }

32 split_on_keyfields (kfs, σ⃗_old, σ⃗_new) =
33   if kfs = [] then return (σ⃗_old, σ⃗_new)
34   σ⃗'_old := []
35   σ⃗'_new := []
36   for each i ∈ 1..length(σ⃗_new) {
37     for each v⃗ ∈ values_kfs(σ⃗_old(i) ∪ σ⃗_new(i)) {
38       σ⃗'_old := σ⃗'_old :: σ_old(i)|_kfs=v⃗
39       σ⃗'_new := σ⃗'_new :: σ_new(i)|_kfs=v⃗
40     }
41   }
42   return (σ⃗_old, σ⃗_new)

43 synthesis_match(σ⃗_old, σ⃗_new) =
44   σ⃗'_old := []
45   σ⃗'_new := []
46   for each k ∈ 1..length(σ⃗_old) {
47    σ := σ⃗_old(k)
48    σ' := σ⃗_new(k)
49    if |σ| ≠ 1 ∨ |σ'| ≠ 1 then
50     while σ' ≠ ∅ {
51       choose o from σ
52       let Δ = ⋃_{o_i∈σ'} synth_non_branching(o,o_i)
53       letfun score(δ) = |σ' ∩ δ(σ)|
54       let δ̂ be the element of Δ that maximizes score
55       let Û = {(o,o') | (o,o') ∈ σ × σ' ∧ o' = δ̂(o)}
56       for each (o,o') ∈ Û {
57         σ⃗'_old := σ⃗'_old :: {o}
58         σ⃗'_new := σ⃗'_new :: {o'}
59         σ := σ − {o}
60         σ' := σ' − {o'}
61       }
62     }
63   }
64   return (σ⃗'_old, σ⃗'_new)
```

Figure 6. Pseudocode for the match function.

Once it scores all the fields, match picks the field g that maximizes the score. If the best score does better at distinguishing objects than it previously did when using just fields in kfs, it adds g to kfs and continues iterating.

If get_fields cannot find a bijection using one or more primitive fields, we extend it by changing the first nested loop (line 12) to consider *field paths*. A field path $\vec{f}$ is a list of fields, e.g. $f_1.f_2$. The value given by $o.\vec{f}$ is the value assigned to field f_2 in the object referenced by $o.f_1$. Beyond the replacement of f with $\vec{f}$, the algorithm in Figure 6 is unchanged. If matching with field paths still does not produce a bijection, we apply synthesis-based matching, described in Section 3.3.

3.2 The Old-Version Consistency Check

Note that the description of matching thus far has not made use of the fact that the objects in the two snapshot lists are the result of executing different program versions. We have described the inputs to match as a list of old-version snapshots and a list of new-version snapshots, but these could equally well be two lists of old-version snapshots produced by separate runs of the old version over a single test case. Performing such an *old-old* match is useful as a preprocessing step prior to *old-new* matching.

Recall that get_keyfields assigns to each field f a score, given by score(f). This score is non-zero if and only if partitioning each snapshot on field f produces sets with equal cardinalities. We use this property in old-new matching as a heuristic to find fields that are unchanged by the update and that help pair up corresponding objects. We would also expect these fields to have non-zero score in an old-old matching, and we use this additional check to ensure that we are in fact finding unchanged fields. In our implementation, we first perform an old-old matching and consider only those fields f with score(f) > 0 as potential key fields for the old-new matching.

Old-old matching also helps focus synthesis. If a field f has score 0 in the old-old case, then this field behaves non-deterministically. As an example, both time-stamps and nonces would have this property. We do not infer transformations for these fields because even if we match objects correctly, the difference between the old and new values for these fields will not solely be the result of the code change.

Finally, if we fail to produce an old-old matching, then we can say with certainty that this is not an object for which we should perform TOS. We can also provide the programmer with feedback indicating that the failure to synthesize is not due to the code change's effect on the object, but rather due to inherent problems with the role of the object in the given test case.

3.3 Synthesis-based matching

If key fields do not induce singleton sets, we further decompose the non-singletons, since whenever $\vec{\sigma}'_{old}(i)$ and $\vec{\sigma}'_{new}(i)$ contain more than one element each, it is unclear which pair (o_1, o_2) with $o_1 \in \vec{\sigma}'_{old}(i)$ and $o_2 \in \vec{\sigma}'_{new}(i)$ to use as an input-output example for synthesis. In this case, match calls synthesis_match with the two current lists of corresponding sets to refine the non-singleton sets in the lists.

Synthesis-based matching tries to find a matching that is witnessed by a transformation function δ that maps objects in the old set to objects in the new set. We introduce additional terminology to explain the algorithm. We say that δ is *consistent* with the object pair (o, o') iff $o' = \delta(o)$. We call a set of pairs U a *matching* iff for all pairs (o_1, o'_1) and (o_2, o'_2) in U, we have $o_1 \neq o_2 \Leftrightarrow o'_1 \neq o'_2$. That is, no old-version object is paired with multiple new-version objects, nor is any new-version object paired with multiple old-version objects.

The synthesis_match function iterates over each pair of object sets, searching for a one-to-one matching between the old and new objects of each when the sets σ and σ' are not singletons. Consider the non-singleton pair σ and σ'. The algorithm first chooses an old-version object o (line 51). It may choose any object since it must eventually find transformation functions for all objects in $\vec{\sigma}_{old}$. It then considers each pair (o, o_i), where o_i is a new-version object, with the aim of synthesizing Δ_i, a *set* of transformation functions consistent with the example (o, o_i). The algorithm combines all these sets of functions into a single set of transformation functions Δ (line 52). The synthesis procedure used here is restricted to non-branching transformation functions. These are functions that do not perform case analysis on the old-version object. (Figure 9 gives pseudocode for this function, which is explained in the next section.) Without this restriction, the inferred functions can always create a conditional function specific only to this one pair, but these functions are not general and do not help create sets of examples for synthesis of general functions.

Line 53 defines a function score that scores possible transformation functions δ in Δ. The algorithm chooses the highest-scoring function $\hat{\delta}$ (line 54). The score of a transformation δ is determined by how well it maps objects in σ to those in σ'—in the figure we write $\delta(\sigma)$ to mean $\{o' \mid o \in \sigma \wedge o' = \delta(o)\}$. Thus a high-scoring function will map the input set to many objects that match (intersect with) the output set. The mapping induced by $\hat{\delta}$ is given in $\hat{U}$. Finally, the loop on line 56 adds the singleton sets comprising this mapping to $\vec{\sigma}'_{old}$ and $\vec{\sigma}'_{new}$ (which will be the ultimate output of this function) and then removes the mapped elements from the current snapshots σ and σ'. The algorithm greedily continues, iteratively choosing transformation functions that maximize the number of example pairs they cover. Once σ becomes empty, which is sure to happen because ultimately objects can be matched arbitrarily using constant functions, the while loop exits and synthesis moves on to the next snapshot, continuing until it considers all pruned snapshots.

4. Synthesis

The synthesis phase takes the example pairs produced by matching and synthesizes a function δ from them such that for the pair (o, o'), $\delta(o) = o'$.

4.1 Transformation functions δ

A transformation function δ is defined according to the grammar in Figure 7. Transformation functions take an old-version object in o, allocate a new-version object in n, assign values to each of the fields of n, and then return n. Field assignments g are of the form $\mathrm{n}.f := c$ where c is a conditional and f is a *field path*—that is, it is a (possibly empty) list of field labels l. For n, this path is almost always a single field. (Section 4.5 discusses the multiple fields case.) Each conditional c specifies one or more cases distinguished by boolean expressions e_i over old-version state (i.e., they can only refer to objects via o, never n). Each case has an initializer expression d which is either a constant, a reference to an old field, a string produced by concatenating one or more (sub)strings, or a collection produced by map. Here, $\mathsf{map}(\delta, \mathrm{o}.f)$ takes the collection at $\mathrm{o}.f$, and transforms these elements using δ.

The string expression $\mathsf{substr}(\mathrm{o}.f, i)$ splits the string $\mathrm{o}.f$ at positions where a delimiter *delim* appears and then selects the i^{th} substring. For example, substr("foo@bar.com", 2) returns "bar.com" since '@' is a delimiter that splits the string into two substrings and "bar.com" is the second substring. Our language of string updates supports a concatenation of substrings and is sufficient for the examples we considered. If a more robust string transformation language were needed, the approach taken by Gulwani [5, 6] or any other example-based string synthesis technique would work easily.

We have simplified the form of δ to keep the algorithm tractable. The most obvious restriction is that δ transforms a single object, rather than multiple objects at once. This restriction derives from the nature of the underlying DSU system we use, Jvolve. Note that different objects of the same

Updates	δ	::=	λo. new n; g_1; …; g_n;**ret** n
Field Updates	g	::=	$n.f := c$
Field Path	f	::=	$\varepsilon \mid f.l$
Conditional	c	::=	case $e_1 \Rightarrow d_1, \ldots, e_n \Rightarrow d_n$ end
Initializer	d	::=	$k \mid \text{o}.f \mid \text{concat}(se_1, \ldots, se_n)$
			$\mid \text{map}(\delta, \text{o}.f)$
Integer Constant	i, j	$\in$	$\mathbb{Z}$
Constant	k	::=	$i \mid$ `null` $\mid$ *delim*
Delimiter	*delim*	::=	`\` $\mid$ `/` $\mid$ `#` $\mid$ `@` $\mid$ `:`
String Expression	*se*	::=	*delim* $\mid \text{substr}(\text{o}.f, i)$
Boolean Expression	e	::=	$a \mid e_1 \wedge e_2 \mid e_1 \vee e_2$
Atomic Expression	a	::=	$\text{o}.f_1 \diamond \text{o}.f_2 \mid \text{o}.f \diamond k$
Operator	$\diamond$	::=	$= \mid \neq \mid \ldots$

Figure 7. The language over which we perform synthesis.

class will not necessarily be transformed in the same way—conditionals c may evaluate to different branches for different objects and thereby trigger different initializers. Another restriction is that each field of the new object is initialized independently, albeit with access to the full contents (i.e., multiple fields) of the old-version object o. Transformers that are ruled out by this setup include those that initialize field f according to the *updated* value for field f' as well as those that pass values across the object graph, e.g., setting $\text{n}.f_2$:= Foo.f_1 when o is not an instance of Foo.

4.2 Synthesis algorithm

Figure 8 gives pseudo-code for the synthesis algorithm. The main function is synthesize. It takes a set of input-output examples U (pairs of old-version, new-version objects) and produces a transformation function that is consistent with those examples.

Synthesis proceeds one field at a time. For each field, it synthesizes a conditional update (c in the grammar in Figure 7) that is capable of producing all the values for that field seen in the example pairs in U. It first generates the initializers d and then searches for conditions that indicate which d to apply.

To find the initializers d, the synthesis algorithm maintains a set update_fns of initializers it has discovered thus far, as well as a set not_covered that contains all example pairs that cannot be produced by an initializer in update_fns. During each iteration through the loop on line 5, it chooses an element from not_covered and calls synth_field, which returns the set D of all initializers d that are consistent with that field's values in the provided example pair. Of these, we choose $\hat{d}$, which covers the largest number of pairs in not_covered, and add it to update_fns, while removing the pairs that $\hat{d}$ covers from not_covered.

The loop on line 16 finds the conditions that indicate which $\hat{d}$ to apply to a given old-version object. For each

```
1  synthesize(U) =
2    for each new-version field f_i {
3      not_covered := U
4      update_fns := []
5      while not_covered ≠ ∅ {
6        choose (o,o') ∈ not_covered
7        let D = synth_field(f_i,o,o')
8        letfun score(d) =
9                  |{(o,o') | (o,o') ∈ not_covered ∧ o'.f_i = d(o)}|
10       let d̂ be the element of D that maximizes score
11       let Û = {(o,o') | (o,o') ∈ not_covered ∧ o'.f_i = d̂(o)}
12       update_fns := update_fns :: (d̂, Û)
13       not_covered := not_covered − Û
14     }
15     cond := []
16     for i ∈ 1..length(update_fns) {
17       let (d̂, Û) = update_fns(i)
18       let σ_in = {o | ∃o'. (o,o') ∈ U ∧ (o,o') ∈ Û}
19       let σ_out = {o | ∃o'. (o,o') ∈ U ∧ (o,o') ∉ Û}
20       cond := cond :: (synth_cond(σ_in, σ_out))
21     }
22     c_fi := ⟨case ĉ_1 ⇒ d̂_1, …, ĉ_n ⇒ d̂_n end⟩
23       where ĉ_j = cond(j) and (d̂_j, Û_j) = update_fns(j)
24   }
25   return ⟨λo. new n; n.f_1 := c_f1; …; n.f_n := c_fn; ret n⟩
```

Figure 8. Main synthesis algorithm.

transformation function $\hat{d}$, the loop builds σ_{in}, containing the old-version objects from the example pairs consistent with $\hat{d}$, and σ_{out}, containing the remaining example pairs. It then calls synth_cond to find a condition that separates these two sets based on values of old-version fields, and adds it to the list cond.

Finally, the algorithm constructs c_{f_i}, the conditional update containing all the logic it just synthesized for field f_i. The update for the class is then the sequence of all these field updates. We write synthesized code in angle brackets to distinguish it from the code of the synthesis algorithm.

4.3 Field synthesis

Figure 9 gives the pseudo-code for synth_field, which produces the initializers for a new-version field according to a given example pair. The figure also shows the code for synth_non_branching, which is the function used to perform synthesis-based matching (Section 3.3). It uses synth_field as a subroutine.

The synth_field function takes a field, an old-version object o, and a new-version object o', and returns a set of initializers (from production d in the grammar in Figure 7), where each initializer can produce the value in $o'.f$ given the object o. The set is constructed by first checking whether the value in $o'.f$ is also present in a field in o and if its value can be copied over. Next, it checks whether $o'.f$ is one of the constants in the synthesis language. If so, it adds this production

```
1  synth_non_branching(o,o′) =
2    g := empty field update
3    for each f in o′ {
4      let c_set = synth_field (f,o,o′)
5      for each c ∈ c_set do { g := g + ⟨ ; n.f := c ⟩ }
6    }
7    return λo. new n; ⟨g⟩;ret n
8
9  synth_field (f,o,o′) =
10   ret_set := ∅
11   if o′.f = o.g for some field g
12     ret_set := ret_set ∪ {⟨o.g⟩}
13   if o′.f = k
14     ret_set := ret_set ∪ {⟨k⟩}
15   if typeof(o′.f) = String
16     ret_set := ret_set ∪ string_synth(o.f,o′.f)
17   if o′.f is a collection
18     let σ′ = multiset of objects in collection o′.f
19     for each collection−valued field o.f2 {
20       let σ = multiset of objects in collection o.f2
21       let δ = collection_synth(σ,σ′)
22       ret_set := ret_set ∪ {⟨map(δ,o.f2)⟩}
23     }
24   return ret_set
25
26 collection_synth (σ,σ′) =
27   let examples = match([σ],[σ′])
28   return synthesize(examples)
```

Figure 9. Non-branching and per-field synth. subroutines.

method to the returned set. Finally, we have two class-based synthesis checks. If $o'.f$ is a string, we invoke string synthesis to produce a set that describes all possible methods for construction the string from substrings present in o. We do not give code for this function since it closely follows Gulwani [5]. If $o'.f$ is a collection, then we recursively invoke synthesis in order to transform the elements of the collection.

4.4 Condition synthesis

Condition synthesis produces a condition that distinguishes two sets of examples following the basic approach of Gulwani [5]. Figure 10 gives the pseudocode. It uses notation $[\![e]\!]$ to denote a function from objects to truth values where $[\![e]\!]\,o = \mathsf{true}$ if and only if condition e is true when o (an actual object) is substituted for o (the variable) in e. If $[\![e]\!]\,o = \mathsf{true}$, we will say that *e includes o* and otherwise we say that *e excludes o*. Given a set of objects σ_{in} and σ_{out}, the goal of condition synthesis is to return an expression e such that $\forall o \in \sigma_{\text{in}}\ [\![e]\!]\,o = \mathsf{true}$ and $\forall o \in \sigma_{\text{out}}\ [\![e]\!]\,o = \mathsf{false}$. If this is the case, we say that *e separates* σ_{in} and σ_{out}.

The construction of e proceeds in a greedy fashion. The loop at line 4 calls synth_conj to synthesize a conjunction $a_1 \wedge \ldots \wedge a_n$, where each a_i is an atomic expression. This expression will exclude all the elements in σ_{out} while including as many elements of σ_{in} as possible. The synth_conj function builds up this conjunction iteratively using the loop at line 15. Given $e = a_1 \wedge \ldots \wedge a_j$, we first check to see if e already separates σ_{in} and σ_{out}. If so, we are done—the code detects this case when $\sigma'_{\text{out}} = \emptyset$. If not, let $\sigma'_{\text{in}} = \{o \mid o \in \sigma_{\text{in}} \wedge [\![e]\!]\,o = \mathsf{true}\}$ and let $\sigma'_{\text{out}} = \{o \mid o \in \sigma_{\text{out}} \wedge [\![e]\!]\,o = \mathsf{true}\}$. Thus σ'_{in} and σ'_{out} are the subsets of σ_{in} and σ_{out} that satisfy e. We then choose a_{j+1} to be the atomic expression that maximizes $\mathit{rank}(a_{j+1}, \sigma'_{\text{in}}, \sigma'_{\text{out}})$. We define the *rank* of a condition e as follows.

Definition 1. *Let* $\mathit{rank}(e, \sigma_{in}, \sigma_{out}) = m \times n$ *where* $m = |\{o \mid o \in \sigma_{in} \wedge [\![e]\!]\,o = \mathsf{true}\}|$ *and* $n = |\{o \mid o \in \sigma_{out} \wedge [\![e]\!]\,o = \mathsf{false}\}|$.

Thus, $\mathit{rank}(e, \sigma_{\text{in}}, \sigma_{\text{out}})$ is the product of the number of objects from σ_{in} that are included by e and the number of objects in σ_{out} that are excluded.

The atomic expressions we consider are those involving equality or inequality between pairs of fields (of which there are a finite number) and equality or inequality with a constant appearing in the object (of which there are a finite number). Since there are a finite number of possible atomic expressions, we can simply iterate over them, although our implementation is able to avoid considering many expressions that are guaranteed to not satisfy the conditions required. Provided $\mathit{rank}(a, \sigma'_{\text{in}}, \sigma'_{\text{out}})$ is non-zero, we know that conjoining a to e will cause some elements of σ'_{out} to be excluded, while still including some elements of σ_{in}. If no atomic expression produces a rank greater than zero, then this indicates a failure to find the necessary condition and we abort the synthesis process for this field. In this case our expression language is either insufficient to distinguish the different examples, or we have simply not considered the discriminating data (e.g., if it were a global variable). Otherwise we continue adding atomic expressions until we have excluded all elements of σ_{out} and constructed conjunct e.

Returning to the code of synth_cond, we know that the conjunct returned from synth_conj excludes all elements of σ_{out}. We add e' to the current list of disjuncts e, and then add to σ'_{in} the set $\{o \mid o \in \sigma_{\text{in}} \wedge [\![e']\!]\,o = \mathsf{true}\}$, which are all objects e now includes. If any elements remain, we iterate again to produce another conjunct that will include them, until the expression includes all elements of σ_{in} (and excludes all elements of σ_{out}).

4.5 Discussion

Our synthesis algorithm is engineered to favor simpler, more general transformations δ over more specialized ones. For example, suppose class Foo contains an integer-valued field f and in all our snapshots $\sigma \in \vec{\sigma}_{old}$ we have two Foo objects o_1 and o_2 such that $o_1.f = 1$ and $o_2.f = 2$. Likewise, all snapshots $\sigma' \in \vec{\sigma}_{new}$ have two Foo objects o'_1 and o'_2 such that $o'_1.f = 1$ and $o'_2.f = 2$. In this case, synthesize will produce the function

$$\lambda \text{o.new n; case } \mathsf{true} \Rightarrow \text{n}.f := \text{o}.f \text{ end; } \textbf{ret } \text{n}$$

```
1  synth_cond(σ_in, σ_out) =
2    σ'_in := ∅
3    e := ⟨false⟩
4    while σ'_in ≠ σ_in {
5      let e' = synth_conj(σ_in − σ'_in, σ_out)
6      e := e + ⟨ ∨e' ⟩
7      σ'_in := σ'_in ∪ {o | o ∈ σ_in ∧ [[e']] o = true}
8    }
9    return e
10
11 synth_conj(σ_in, σ_out) =
12   σ'_in := σ_in
13   σ'_out := σ_out
14   e := ⟨true⟩
15   while σ'_out ≠ ∅ {
16     let a be the condition that maximizes rank(a, σ'_in, σ'_out)
17     if rank(a, σ'_in, σ'_out) = 0 then abort
18     e := e + ⟨ ∧a ⟩
19     σ'_in := {o | o ∈ σ'_in ∧ [[e]] o = true}
20     σ'_out := {o | o ∈ σ'_out ∧ [[e]] o = true}
21   }
22   return e
```

Figure 10. Synthesizing conditions.

and not the function

$$\lambda \text{o.new n;case } (\text{o}.f = 1) \Rightarrow \text{n}.f := 1, \quad \text{case } (\text{o}.f = 2) \Rightarrow \text{n}.f := 2 \text{ end; ret n}$$

Line 10 of synthesize (Figure 8) favors initializers that apply to the most possible objects. Line 16 of synth_conj (Figure 10) similarly aims for simpler, more general conditions.

The algorithm as described assumes that synthesis takes place on a per-object basis, one field at a time. In fact, it can also synthesize the contents of a new object's children, e.g., assigning not just $\text{n}.f := d$ but $\text{n}.f_1.f_2 := d$ as well. Likewise it can read children of old objects in initializers d. To support this extension, we simply consider field *paths* rather than fields, up to a specified depth, e.g., on line 2 in Figure 8 and on line 11 in Figure 9. Matching can be similarly extended, e.g., using paths on line 12 in Figure 6. Finally, we extend the synthesis language in Figure 7 to allow object allocation, so we can allocate and initialize child objects if needed. (Note that allocation happens implicitly with map when producing the new collection.) We have found this flexibility useful in practice, as we discuss in the Azureus example in the next section.

5. Evaluation

We implemented TOS and used it to synthesize object transformers for several program updates we gathered from the wild. We choose challenging examples that other systems cannot handle, and find that TOS handles most. TOS does fail when the changed objects do not store data that makes it possible to match them between heaps, but this case is less common. This section provides a few more details about our implementation, describes each test program update, and our experiences with TOS in each case.

5.1 Implementation

We use the Oracle HotSpot JVM to collect heap snapshots using the `agentlib:hprof` command line option.[2] This option invokes the heap profiler, which sends the current snapshot over a network socket. We wrote a small server that coordinates snapshots with the application. It initiates snapshots at update points and saves them to disk. In particular, when the application reaches an update point, it calls into a helper class to trigger a snapshot.

TOS is implemented in Java and comprises roughly 4200 lines of code. About 1300 lines implement matching, 1600 implement synthesis, and the rest is common code.

5.2 Results

In our experiments, we consider two updates to Azureus (a bittorrent client); three updates to jEdit (a text editor); one update to JavaEmailServer (a.k.a., JES, an SMTP and POP mail server); and one update to CrossFTP (an FTP server). These applications range from 2.3k to 250k source lines of code. These are actively developed and maintained programs not built with dynamic software updating in mind.

We chose these applications and updates for the following reasons. First, dynamic updates for these programs would be useful. For Azureus, JES, and CrossFTP, dynamic updates would avoid disruption due to shutdown/restart, e.g., they would preserve active connections involving e-mail sending or file transfer, and they would preserve important in-memory state, such as file/metadata caches. Dynamic updates to JEdit are less important (you could save in-progress work and restart), but would be convenient, e.g., to preserve the content and layout of onscreen windows. Second, the updates to JES and CrossFTP were also considered in the original Jvolve work [14]. We wanted to see if we could use TOS to generate transformers we previously wrote by hand. Finally, we found updates to these programs that are relatively interesting, such as updates that do not affect all objects uniformly (thus requiring conditional transformers) or updates that fix memory leaks. We focus on the challenging updates that prior DSU systems cannot handle automatically. We also tested TOS on many simple updates, and include two of these—the "copy" transformers for CrossFTP v1.06—as exemplars.

Table 1 summarizes information about the updates and the inferred transformers. We list the version for which we synthesize an update and the number of snapshots generated by our test case. In all but one example, TOS needs only a single test case. Version 1.08 of CrossFTP required three test cases. We list the number of classes used by the pro-

[2] We use HotSpot only for snapshotting—Jvolve, the VM that TOS targets, is based on Jikes RVM and does not support snapshotting.

Application	SLOC	Version	# Snaps	Classes	Heap Obj.	Target Obj.	Match	Synthesis	Inferred	Type
Azureus	250K	r2514	11	1616	1315420	97	0.842 s	0.120 s	yes	conditional
		r120	22	1634	1117463	275	0.002 s	0.000 s	no	
JEdit	154K	r14027	5	3044	703360	30	0.041 s	0.008 s	yes	constant
	150K	r13413	5	3221	747849	0	0.000 s	0.000 s	no	
JES	2.3K	1.3.2	1	911	51802	1	0.020 s	0.022 s	yes	collection
	2.4k	1.3.3	1	902	52210	1	0.001 s	0.007 s	yes	constant
CrossFTP	13.9K	1.06	1	953	38907	1	0.001 s	0.009 s	yes	copy[1]
	13.9K	1.06	1	953	38907	1	0.002 s	0.007 s	yes	copy[1]
	13.9K	1.06	1	953	38907	1	0.002 s	0.009 s	yes	constant
	14K	1.07	1	2417	152531	1	0.044 s	0.013 s	yes	constant
	18.1K	1.08	3	954	116833	1	0.002 s	0.011 s	yes	conditional

Table 1. Summary of updates and inferred transformers. Each example is the result of a single test case. [1]Transformer that would also have been produced automatically by Jvolve.

gram, the number of object instances present (total across all snapshots), and the total number of instances of the target object (target objects changed or transitively referred to by changed objects). Finally, we list execution times in seconds for matching and synthesis, and indicate if synthesis succeeded.

Matching and synthesis times are negligible for all examples. Matching times and effectiveness benefit from focusing on a single class at a time, since it significantly reduces the number of objects that TOS considers. TOS restricts itself to changed objects and objects reachable from changed objects by a bounded number of field dereferences. This bound is the same one mentioned in Section 4.5, which discusses bounding the depth of field paths. Synthesis benefits from the fact that our language of state transformers is designed to be small and succinct.

The synthesized functions involve constant updates, conditional updates, string transformations, and collection updates. For JES and CrossFTP, we used the generated update functions in Jvolve [14] and verified that the system continued running correctly following the update. We were unable to get Azureus and jEdit to run reliably using the Jvolve VM, but this was not due to Jvolve: the Jikes RVM, on which Jvolve is based, does not run them properly either. Since we could not run the Azureus and jEdit updates, we performed a manual code review of the transformers produced by our synthesis algorithm to check that they matched what we would have written by hand.

TOS fails in two cases to synthesize update functions because we could not generate snapshots that captured the changed behavior; we discuss these situations, and all of the updates, in detail below.

Azureus update SVN r2514 Azureus is a widely used BitTorrent server/client. Version r2514 contains about 250k lines of code. This update changes two classes and methods. Figure 11 shows a portion of the update. The added call to clearServerAdapter nulls the adapter field of the _server object when a peer server is stopped; doing so ensures the adapter is garbage collected. When we apply this update at run-time, we would like it to retroactively null adapter fields that the buggy version neglected to. TOS facilities doing so by synthesizing a transformer for PEPeerControlImpl objects.

To generate the snapshots, we ran both program versions in a controlled setting and had them download the same set of files. TOS performs matching on PEPeerControlImpl objects. The program contains one such object for each file it downloads. Matching chooses key field _nbPieces, which is a value proportional to the file size that is highly likely to be unique across different files.

Synthesis then observes that for these matching objects the new versions' _server.adapter field is **null** when _server.bContinue is false, but the two versions match on the _server.adapter field otherwise. It then infers the following conditional transformer, which has the effect of freeing the leaked objects:

```
if (_bContinue == false)
  _server.adapter = null;
else
  _server.adapter = _server_old.adapter;
```

Note that to generate this transformation function requires using field paths instead of single fields (cf. Section 4.5).

Azureus update SVN r120 Figure 12 shows update r120 to Azureus. This update modifies the condition under which Azureus releases the read buffer between a client and its peer. There are two issues with using TOS to infer this update. The first is that the read buffer does not contain a natural key field. Each buffer is identical and does not have a pointer back to the object using it. This problem manifests as a failure in the matching process. In such a case, one might consider adding a ghost field that records the allocation order and using this as a key field to match objects. Matching succeeds with this addition. However, the assignment of

```
class PEPeerControlImpl {
  PESharedPortServerImpl _server;
  boolean _bContinue; ...
  void stopAll () { ...
    //3. Stop the server
    _server.stopServer();
+   _server.clearServerAdapter();
    ... }
}
class PESharedPortServerImpl {
  void clearServerAdapter() {
    adapter = null;
  }
}
```

Figure 11. Azureus r2514 update

```
public class PeerSocket ... { ...
      //4. release the read Buffer
-     if (readBuffer != null && !readingLength)
+     if (readBuffer != null)
        ByteBufferPool.getInstance().freeBuffer(readBuffer);
 ... }
```

Figure 12. Azureus update r120

buffers to clients varies across runs (and is unrelated to allocation order), which causes synthesis to fail, since the i^{th} buffer is not associated with the same client in each run. This transformation is thus not inferable from the objects themselves, which is a limitation of our approach. However, it is not surprising that, on occasion, sufficient information for transformation is not available from the heap.

jEdit JEdit is a text editor for programmers that provides common and advanced features, such as syntax highlighting, folding, automatic indentation, a built-in macro language, macro recording, and plugin support. Here we consider two similar memory leaks fixed in jEdit versions r5178 and r14027. Figure 13 shows the source patch for the leak fixed in r14027. jEdit calls the function markTokens when it needs to split a string into tokens based on the type of the file being edited. Each file type (C, Java, Verilog, etc.) has special logic to split text in a line and embed it in an object of type TokenHandler. The leaky jEdit version fails to set the field TokenMarker.tokenHandler to **null**.

The inferred object transformer is simple. It sets the leaky field to **null**. It is safe to execute the transformer as long as the markTokens function is not active on stack.

Figure 14 shows a change to jEdit in SVN revision r13413. The leak is in function HistoryText.showPopupMenu(). jEdit calls showPopupMenu() when the user performs certain actions, such as right clicking a text field with history. In the old version, some instances of HistoryText have the boolean field popup.visible set to true and others set it to false. In the new version, the field popup is not null only when an object also has its popup.visible field set to true. TOS can infer

```
class TokenMarker {
  public LineContext markTokens(...) {
        ...
        tokenHandler.setLineContext(context);

        /* for GC. */
+       this.tokenHandler = null;
        this.line = null;
        return context;
    }
}
```

Figure 13. Update r14027 to jEdit

```
class HistoryText {
 showPopupMenu() {
   if (popup != null && popup.isVisible())
   {
       popup.setVisible(false);
+      popup = null;
       return;
   }

- popup = new JPopupMenu();
+ popup = new JPopupMenu() {
+      @Override
+      public void setVisible(boolean b) {
+          if (!b) {
+              popup = null;
+          }
+          super.setVisible(b);
+      }
+ };
   JMenuItem caption = new JMenuItem(jEdit.getProperty(
       "history.caption"));
   caption.addActionListener(new ActionListener()
 }
}
```

Figure 14. Update to jEdit: SVN revision r13413

```
class DataConnectionConfig {
    ...
+ boolean enableBonjour = true;
+ String listEncoding = "utf-8";
}
```

Figure 15. Update to CrossFTP in version 1.07

this property and generate a transformer that nulls the popup field of instances that have popup.visible set to false. However, while creating snapshots we were unable to create a situation where popup.isVisible() was true during a snapshot, which would execute instructions controlled by the condition and exercise the leak. This update is an example that TOS is capable of handling, but for which we fail due to test coverage problems.

JavaEmailServer (JES) The collection update is the one discussed in Section 2 and presented in Figure 3. We automatically generate a correct update function for it.

The constant update involves the addition of a new field deliveryAttemptThreshold. This value controls how many times JES should try to deliver a message before discarding the message. We automatically generate an initializer that sets this to the default value. This default value is not present in the code, but rather in a configuration file that is read during JES start-up. Thus it would not have been discovered by other approaches to transformer generation.

CrossFTP In this Java-based FTP server, the DataConnectionConfig class maintains configuration information about the connection between the client and the server. Its fields control what response the server sends to various commands a client issues. Version 1.07 of CrossFTP adds two new fields to this class. The boolean field enableBonjour controls whether the server should support the Bonjour protocol and the String field listEncoding specifies what encoding the server should use when responding to the LIST command. There is always only one instance of this object in the heap. From the heap snapshots, TOS identifies the value of these fields in the new version and generates the transformation function that sets these fields accordingly.

For version 1.06, we consider three field updates, two of which involve copying the old version to the new and one of which involves a constant initializer. The copy cases involve fields whose access modifiers have changed. In such cases, simply copying the old value to the new is a good heuristic and this transformer would be generated automatically by Jvolve. Our system also generates the transformer but provides the added assurance that this update is consistent with the states observed when running the old and new versions.

In version 1.08, the default port used by a SocketFactory object is changed. We provide TOS with three snapshots at each version—two in which non-default ports are configured and one that has no port configured and thus falls back on the default port. TOS is able to synthesize the conditional transformer that changes the port to the new default value if it was set to the old default value and leaves it unchanged otherwise.

Discussion These first experiments with TOS show that good results can be obtained when matching and synthesis work in harmony. If either step fails then TOS fails for the targeted class. But if matching identifies key fields then synthesis-based matching can be avoided or reduced and low TOS run times are observed. For the examples we considered, most objects do have key fields that matching identifies and uses to produce good examples for synthesis. Although it seems intuitive that most programs will encode sufficient state in changed classes to make matching practical, future work should explore this question more thoroughly. Our synthesis language is relatively simple, which eases synthesis, yet it includes common string and data functions. Future work should explore if the current synthesis language has sufficient coverage on a wider range of programs.

6. Related work

This paper contributes novel matching and synthesis algorithms. The matching algorithm analyzes unstructured heap snapshots from different program version executions. While some recent prior work analyzes a single heap to discover leaked objects and other inefficiencies [2, 3, 9, 11, 12, 15], none aligns heap objects from different program versions or considers how to fix the effects of leaks on the fly.

A lot of related work considers synthesizing code from specifications, but only recently have researchers considered the problem of synthesizing data transformation functions. The closest related work is by Gulwani and others on synthesizing string and Excel spreadsheet data transformations. These approaches require users to directly specify the input/output examples whereas TOS requires users to run the same test on both program versions from which examples are inferred by matching. Gulwani's algorithm [5] synthesizes string functions that include concatenation, subsequence, and finding special symbols. TOS uses this algorithm as a subroutine as part of synthesizing transformations between objects (cf. Section 4.3). Harris and Gulwani [6] generate transformations between spreadsheets; their numeric transformations and filters are similar to ours. They also search structured spreadsheet data to find correlation between the input and output rows and columns. Our matching phase serves a similar purpose, but once objects are paired up, synthesis follows the structure of the new-version object, assigning its fields one at a time. A unique feature of TOS is that it iterates synthesis and matching to produce transformers for collections of objects.

Many prior dynamic updating systems, including Ginseng [13], DLpop [8], POLUS [4], and Jvolve [14], provide primitive support for generating state transformation code. For changes that extend classes or structs with new fields, these systems simply copy the old fields and initialize the new ones with default values, e.g., **null** for object references, or 0 for **int**s. Systems that provide no direct support for state transformation, e.g., LiveRebel [16], effectively take this approach. In all of these cases, synthesis is based entirely on comparing the definitions of changed types/classes. None of them consider program semantics by analyzing the code or the heap. By contrast, TOS obtains semantic information from program execution to derive data transformations. In short, while prior systems remove some of the tedium of writing transformation functions, they fail to handle any interesting program changes, which are exactly the cases which are harder for programmers to write correctly.

7. Conclusions

This paper has presented *Targeted Object Synthesis* (TOS), a novel technique that synthesizes *object transformer methods*. Object transformers convert old version objects to new ones during a dynamic software update. TOS is distinguished by its generality: whereas prior techniques for syn-

thesizing object transformers follow simple syntactic rules, TOS produces functions based on observations of actual program executions of the old and new program versions. In particular, TOS takes periodic heap snapshots at corresponding points during executions of the old and new program when executing the same inputs. It then *matches* corresponding objects between these snapshots, and uses these as examples to *synthesize* object transformation functions. We show TOS is efficacious in synthesizing transformation functions for actual changes to classes in various Java server applications. Even when it fails to generate a correct transformer, the partial results may be useful to developers. This functionality eases, but does not eliminate, the programmer burden of understanding program changes and performing dynamic software updating. TOS may also be useful for other version and program understanding scenarios, such as bug detection and testing.

Acknowledgments We thank the anonymous reviewers for helpful comments on drafts of this paper. This work is supported by NSF grants CCF-0910530, CCF-1018271 and SHF-0910818 and the partnership between UMIACS and the Laboratory for Telecommunication Sciences. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] J. Arnold and F. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *EuroSys*, 2009.

[2] M. D. Bond and K. S. McKinley. Bell: Bit-Encoding Online Memory Leak Detection. In *ASPLOS*, 2006.

[3] M. D. Bond and K. S. McKinley. Leak pruning. In *ASPLOS*, 2009.

[4] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. POLUS: A POwerful Live Updating System. In *ICSE*, 2007.

[5] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[6] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, 2011.

[7] C. M. Hayden, E. K. Smith, E. A. Hardisty, M. Hicks, and J. S. Foster. Evaluating dynamic software update safety using efficient systematic testing. *IEEE Transactions on Software Engineering*, 99(PrePrints), Sept. 2011.

[8] M. Hicks and S. M. Nettles. Dynamic Software Updating. *Transactions on Programming Languages and Systems*, 27(6): 1049–1096, November 2005.

[9] M. Jump and K. S. McKinley. Cork: Dynamic Memory Leak Detection for Java. In *POPL*, 2007.

[10] K. Makris and R. Bazzi. Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. In *USENIX ATC*, 2009.

[11] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *OOPSLA*, 2007.

[12] N. Mitchell and G. Sevitzky. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In *ECOOP*, 2003.

[13] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical Dynamic Software Updating for C. In *PLDI*, 2006.

[14] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic Software Updates: A VM-centric Approach. In *PLDI*, 2009.

[15] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *PLDI*, 2010.

[16] ZeroTurnaround. LiveRebel. `http://www.zeroturnaround.com/liverebel`.